Plagiarism in open source software projects: which developers are more prone to code cloning?

N. van Bremen - 5953790 - j.a.vanbremen@students.uu.nl
J. Geel - 5932432 - j.t.geel@uu.nl
N.J. de Rijk - 5670721 - n.j.derijk@students.uu.nl

Utrecht University, Department of Information and Computing Sciences, 3584 CC Utrecht, Princetonplein 5, science.secr.cs@uu.nl, https://www.uu.nl/

Abstract. Plagiarism is often considered a violation of the academic integrity or the breach of journalistic ethics. However, with the rise of the Internet and the subsequent facilitation of sharing source code with one another, this has opened the door for another form of plagiarism, namely code cloning. Previous research has indicated that code cloning occurs often in large software ecosystems and can be considered bad practice. However, other research claims that code cloning is not as bad as some say. It is not known whether a particular developer type is more prone to code cloning than other developer types. To find out whether a particular developer type is more prone to code cloning than others, methods in open-source software projects have been analyzed. Consequently, developers were categorized into types, namely Networkers, Lone Wolves, and One Day Flies. Additionally, developer experience is measured. Over 8.8 million methods written by over 46 thousand developers in 29,000 thousand projects were analyzed. The analysis shows that Networkers are more prone to cloning than Lone Wolves and One Day Flies. It also shows that developers with high development experience are more prone to cloning than developers with lower development experience.

Keywords: Code Cloning · Developer Types · Vulnerabilities · Developers · Open source · Software Projects · Software Ecosystems · Plagiarism · SecureSECO

1 Introduction

Code cloning is a common occurrence in software development [1] and has been a growing problem for the last few years [5, 15]. However, this problem has gone largely unstudied for corporations. Code cloning can yield problems such as developers not understanding the source material including unlicensed content, and introducing vulnerabilities that were already present in the source code [9, 2, 11, 6]. In a study performed by the Synopsys corporation in 2021, it was found that 84 percent of 1.500 codebases that were audited contained at least one public

open source vulnerability [20]. Of these code bases, 60 percent contained high-risk open source vulnerabilities. Vulnerabilities in software can not only result in additional costs for an organization, but also in damages to the organisation itself [8]. Furthermore, when considering open-source software, vulnerabilities therein also have consequences for the products that use them [17].

However, some studies suggest that code cloning is not always bad practice [11]. An example: "Cloning is also used to separate the dependencies of custom views on data that several modules or applications may have. Cloning in this way prevents the introduction of bugs into working code, and confines testing to a smaller subset of source code."[4].

If there is a relation between code cloning and developer type, it could be interesting for organizations that are hiring developers to know beforehand what type of developer the applicant is, so these organizations can get better insights into potential code cloning. However, it is up to the organization to pass judgement on whether code cloning is a particularly bad or a good thing.

For these reasons, this study focuses on investigating the relationship between code cloning and developer types. In the context of this study, code cloning is defined as a developer fully cloning a programmatic method into a new software project. Additionally, this study investigates code cloning in the context of open-source projects that can be found on GitHub. The question that guides this research is defined as follows:

> *What relationship exists between the number of methods a developer clones and the developer type who introduces these methods in open-source projects?*

This study has the following contributions. Section 2 introduces the research method. Section 3 gives some background of the current literature. Section 4 presents the results of the study. Section 5 discusses the results of the study. Finally, Section 6 presents the overall conclusions of the study and presents suggestions for future work.

2 Research method

2.1 Goals and hypotheses

As stated in Section 1, the main question of this research is whether there is a relationship between the number of methods a developer clones and the developer type that introduces these methods in open-source projects. This research question will be tested with the following hypotheses:

Null Hypothesis (H_0): There is no statistically significant relationship between the number of methods a developer clones and the developer type who introduces these methods in open-source projects.

Alternative Hypothesis (H_A): There is a statistically significant relationship between the number of methods a developer clones and the developer type who introduces these methods in open-source projects.

To answer the research question and test the hypotheses, two aspects of the process of writing software are taken into consideration. First, the experience of the developer is examined, because it could be that inexperienced developers clone more code than experienced developers. Second, the type of developer is examined, because it could be that a type of developer who writes most of the code relative to its team members also clones more code than its team members. Both aspects are measured independently of each other. The following gives an overview of the sub-questions to answer the main research question:

- **Sub-question 1.1 ($RQ1.1$):** Does the experience of the developer influence the number of code cloned?

- **Sub-question 1.2 ($RQ1.2$):** What groups of developers can be distinguished within the open-source ecosystem?

2.2 Data collection

For this research, the SearchSECO project is used. The database of SearchSECO contains methods of all kinds of open source projects from GitHub. These methods are stored in such a way that it is possible to compare them and find which methods are clones of other methods.

To get the information from the database, the database API had to be used[1]. In order to do this, a list of 123,609 authors was acquired and used to get all methods every author had worked on. From this list, eventually 46,560 authors were used. This was mainly due to the amount of data gathered, as only gathering the methods of 46 thousand authors already yielded in a data set with over 30 million methods. This dataset, consisting of methods and the corresponding author, was used to deduct other information from.

To determine the developer type, data on the popularity of the projects was also needed. The earlier mentioned data set contained a row for every method written by an author. This row also contained the project ID the method was found in and the project version. In total, the methods were found in 29,774 projects. This information was used to query the database to also give the names of the projects. This name was then used to analyse the popularity of a project based on hits on Yahoo. Table 1 summarizes the data gathered, which research question it aims to answer and for which variable it was used. Table 1 also shows three different developer types: Networker, One Day Fly and Lone Wolf. These types have been adapted from [10] and will be explained further in the following section.

[1] SearchSECO Database API: https://github.com/SecureSECO/SearchSECODatabaseAPI

Unit	Research Question	Used for
Number of methods written	1	Developer experience
Number of projects contributed to	1, 2	Developer experience, networker type
Number of cloned methods	1	Cloned methods
Number of Yahoo hits per project name	2	One day fly type

Table 1: Data collection

2.3 Calculating developer experience

In order to calculate developer experience (RQ1), and as seen in Table 1, we look at number of methods written and number of projects contributed to. Every developer can score from 1.0 to 5.0 in both categories. When a developer has written the most amount of methods and contributed to the most projects, the developer scores 5.0 in both categories. If the inverse were true, the developer would score 1.0 in both categories. The corresponding scores are defined by looking at the distribution of the data. If there are many developers who have only written a minimal amount of methods, this bin would become smaller to account for the loss in other bins. When all data points are placed in the appropriate bins, the average of these two scores is taken and a label is given regarding the developer experience. The higher scores will receive the 'High experience' label, the lower scores will receive the 'Low experience' label and the average scores will receive the 'Medium experience' label. To account for variance of the data, the labels are given with respect to the amount of developers that have scored a certain amount. When there are many developers who do not receive a very high score, the bin for the 'High experience' label becomes higher.

3 Background

It was found that large portions of code are cloned within big software systems [13], even though research suggests that copy-pasting code is a bad practice that may introduce problems. Especially the maintainability of a project suffers when a lot of code is copied [9], although some studies contradict this and claim that duplicate code is not as harmful as is often suggested [2, 11]. Code cloning may even have certain benefits. However, in general the benefits do not seem to outweigh the potential disadvantages [12].

Code cloning does not only occur within projects, but also between projects. Cloning code between projects may bring different kinds of problems. There may for instance be legal implications when cloning code, depending on the terms of the license of a piece of code, although this does not seem to give a lot of issues [6]. A more pressing problem is the introduction of vulnerabilities in

software because of cloned code [16]. The detection of cloned code has become of great scientific interest because of these potential harms and many tools and techniques for clone detection have been proposed [18].

There are different reasons for a programmer to clone code. In an extensive review on software clone detection, it was found that code may be copied due to e.g. time constraints, limitation of programmer's skill, and programming language limitations [18].

As mentioned above, the (lack of) skill of a programmer may influence their cloning behaviour. But code cloning is not only done by novice programmers, experts may also clone code into their projects [3, 7]. An important difference may be the comprehension of the copied code, as a programmer does not always understand what the code they have copied into their project actually does [14]. When a developer does not fully grasp the meaning of a cloned fragment or its origin, they may overlook the implications of cloning it into a project.

4 Results

This section answers the research questions based on quantitative data.

4.1 Data preparation

After the data had been collected as described in Section 2, the data was prepared for analysis. The data set with all the methods written by one author was used to determine how many methods were written by an author and in how many projects they participated. It was found that for many methods, there were multiple duplicates of that method in different versions of the same project. As this should not count as an additional method written, superfluous combinations of the author, method, and project were filtered out.

It was then found that when a project was forked, a method in the new project would be attributed to the author of the original project from which the new project was forked. This should not count as an extra method written by an author nor as an extra project contributed to. These methods could be filtered by removing duplicates of the combination of author, method, and project version, as the projects these methods were in would always differ, but the version and author would be the same. After filtering the data on different versions of the same method and methods in forked projects, a data set with 8.8 million methods remained.

The number of methods each author had cloned was determined using the filtered data set as described above. To get the clones of a method, it was important to first establish which of the clones had been written first. If the first method had multiple authors, these would all be excluded from the list. The authors of the other clones would receive a plus one which would eventually lead to the total number of methods cloned by each author. To get a better idea of how often an author cloned, the number of clones was divided by the number of methods written to determine the relative number of methods cloned per author.

After all duplicates were removed, the data set was inspected for outliers. Initially, the amount of authors investigated were 47,540. After calculating the amount of methods written and projects contributed to for these developers, box plots were created for these metrics to illustrate the data's spread and skewness. These box plots can be observed in Figures 1 and 2.

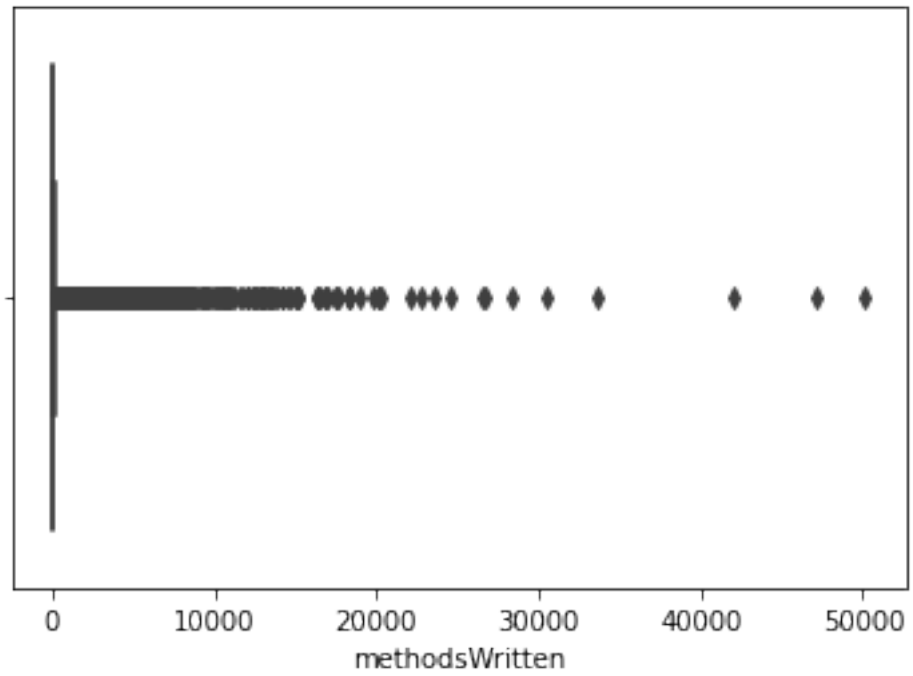

Fig. 1: Boxplot of the amount of methods written by authors before removing outliers

As can be seen in Figures 1 and 2, the amount of outliers (i.e. the individual points beyond the whiskers of the plot) are substantial. To reduce the influence of these outliers on the data, it was decided to filter out the following amounts of methods written and project contributions respectively:

Fewer than 5 and more than 200 methods written. Fewer than 2 and more than 10 projects contributed to. The removal of these data points resulted in the boxplots illustrated in Figures 3 and 4.

After removing the outliers, the amount of developers in the data was reduced from 47,540 to 9,135.

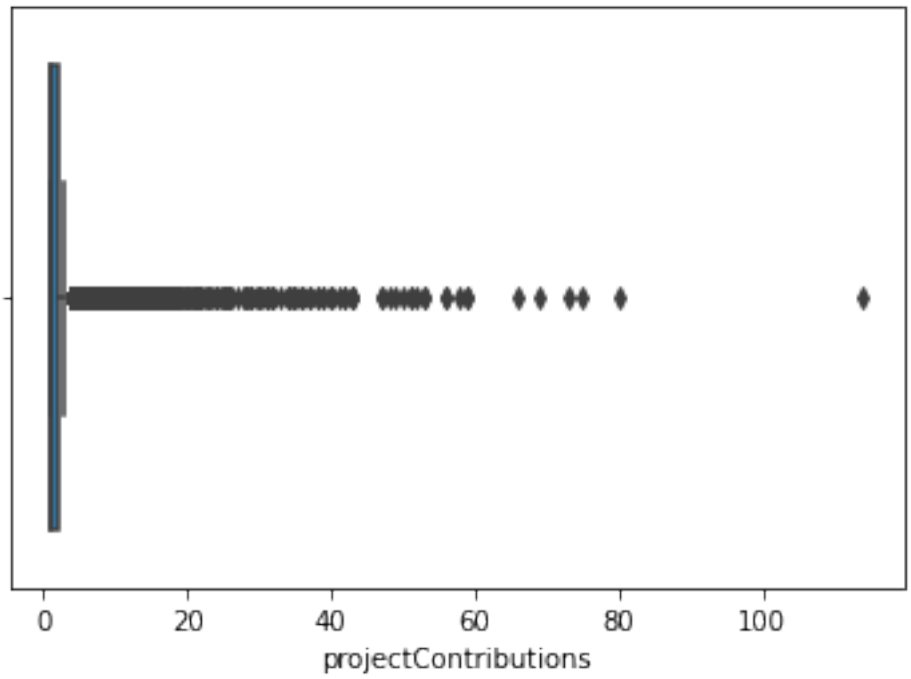

Fig. 2: Boxplots of the amount of projects contributed to by authors before removing outliers

4.2 Measured variables

Developer experience After outliers had been removed from the data, the distribution of project contributions and methods written per developer was as illustrated in Figure 5.

As can be seen in Figure 5, the height of the bars indicates the central tendency for the amount of methods written per amount of project contributed to by developer. Also, error bars have been added to the bars to indicate the degree of uncertainty.

Based on these values and the methodology presented in Section 2, the following bins were defined for projects contributed to and methods written respectively to categorize the developer's experience.

For methods written, five bins were defined: [0 - 15], [15 - 25], [25 - 55], [55 - 75], [75 - Infinity].

For projects contributed to, five bins were defined: [0 - 2], [2 - 3], [3 - 5], [5 - 7], and [7 - Infinity].

After averaging the developer's scores for both bins, the distribution of scores became visible. Based on these scores, the developer experience categories were

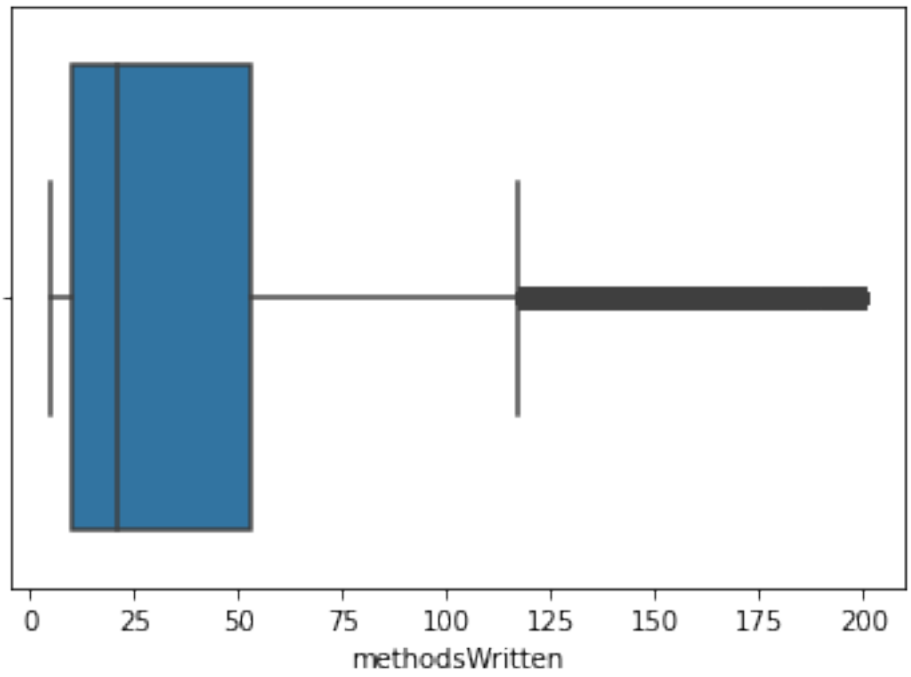

Fig. 3: Boxplot of the amount of methods written by authors after removing outliers

defined. These scores and the corresponding categories can be observed in Table 2.

Score	Developer count	Developer experience
1.0	1,759	Low experience
1.5	1,355	Medium experience
2.0	1,656	Medium experience
2.5	1,180	Medium experience
3.0	1,481	High experience
3.5	817	High experience
4.0	525	High experience
4.5	230	High experience
5.0	132	High experience

Table 2: Developer experience scores, the corresponding amount of developers attributed to the scores, and the associated developer experience rank

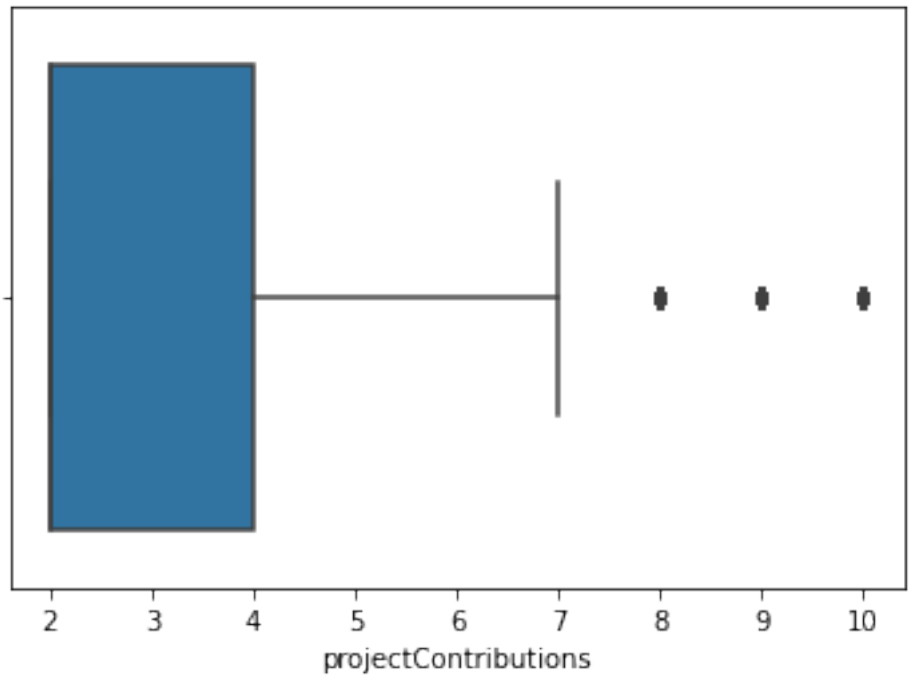

Fig. 4: Boxplots of the amount of projects contributed to by authors after removing outliers

The resulting distribution of experience of developers can be observed in Table 2.

Developer experience	Number of developers
Low experience	3,185
Medium experience	3,114
High experience	2,836

Table 3: Developer experience groups

Unsupervised clustering with K-Means As described in Section 2, an unsupervised clustering algorithm is used to discover clusters in the data. On the remaining 9,135 developer the K-Means algorithm will be applied. First, to observe to optimal amount of clusters for these data, an elbow test is performed [19]. The code for this can be observed in Appendix A. The output of this test can be observed in Figure 6.

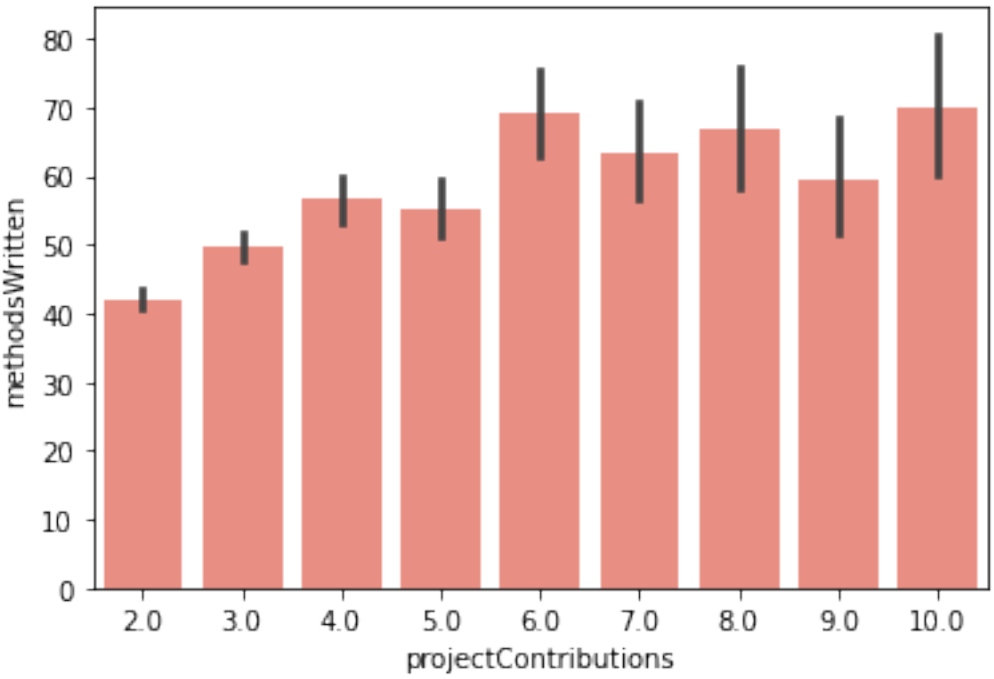

Fig. 5: Barplot of amount of projects contributed to and methods written per developer with outliers removed

The interpretation of this test is as follows: the optimal amount of clusters can be indicated by the the point of inflection on the curve (i.e. the 'elbow') shown by the line plot. The distortion, on the left y-axis, is a representation of the Sum of Squared Errors (SSE). The SSE is calculated for each number of clusters k and the goal is to minimize this. However, as k increases, the SSE goes down by default. Therefore, the goal is to choose a small value of k for the optimal SSE (i.e. the point where diminishing returns take effect) and in some cases the optimal fitting time for the K-Means model (i.e. the green line corresponding to the right y-axis). Since fitting time is not a concern for this model, only the optimal amount of clusters k is considered. In the case of Figure 6, the optimal point of SSE is at $k == 4$.

Table 4 highlights the output of groups after applying the K-Means algorithm at k=4.

Afterwards, the data of these groups were inspected to label them. This was done by inspecting the mean and median values of all data points for developers in their respective groups. Table 5 highlights the results for the mean and Table 6 highlights the results for the median.

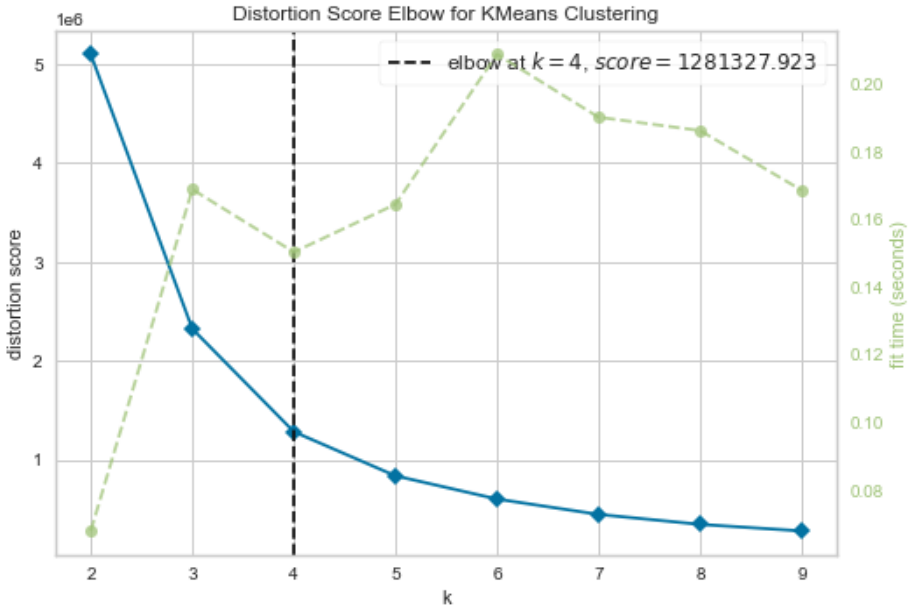

Fig. 6: Line plot showing the amount of clusters and the respective distortion

K-Means group	Amount of developers
0	4,989
1	2,161
2	1,222
3	762

Table 4: Overview of assigned groups by K-Means at *k=4*

K-Means group	Projects contributed to	Methods written	Relative clones	Avg. search results of projects
0	2.84	12.90	0.27	19,685,960
1	3.48	169.63	0.33	15,013,900
2	3.40	71.50	0,28	23,358,550
3	3.59	115.53	0.29	21,135,360

Table 5: Mean values of assigned groups by K-Means at *k=4*

K-Means group	Projects contributed to	Methods written	Relative clones	Avg. search results of projects
0	2	12	0.2	3,419,216
1	3	168	0.28	2,827,000
2	3	70	0.22	3,337,396
3	3	115	0.23	3,539,688

Table 6: Median values of assigned groups by K-Means at *k=4*

4.3 Statistical tests

To analyze the relationship between the assigned developer group by K-Means & experience, and code cloning, a statistical test needs to be performed. Since the developer group & experience are categorical variables and code cloning is expressed as a continuous variable, two Kruskal-Wallis H-tests are performed. One between the assigned K-Means groups and the (relative) code cloning. And another on between the developer experience and the (relative) code cloning.

Developer experience and code cloning First, a table was created highlighting the statistics of the considered variables. This information can be observed in Table 7.

Developer Experience	N	Mean	SD	SE	95% Conf. Interval
High	3,185	0.3249	0.2786	0.0049	[0.3152 0.3346]
Medium	3,114	0.2189	0.2304	0.0041	[0.2108 0.2269]
Low	2,836	0.2673	0.2621	0.0049	[0.2576 0.2769]

Table 7: Table displaying the statistics of relative number of code cloning per developer experience group

Second, assumptions about the data must be checked before performing the Kruskal-Wallis H-test.

The assumptions are as follows:

- One independent variable with two or more levels (i.e. independent groups).
- The dependent variable is of an ordinal, ratio, or interval scale.
- All groups should have roughly the same shape distributions.

In the case of this project's data, there is indeed one independent variable (i.e. developer experience) that has three groups. Furthermore, the dependent

variable (i.e. relative amount of times a developer has cloned code) has an interval scale. Lastly, the groups of developer types have roughly the same shape distributions.

The overall average code cloning was 0.270 95% CI(0.2612, 0.279) with group averages of 0.3249 95% CI(0.3152, 0.3346) for the high developer experience group; 0.2189 95% CI(0.2108, 0.2269) for the medium developer experience group; and 0.2673 95% CI(0.2576, 0.2769) for the low developer experience group.

There is a statistically significant difference between the developer experience groups and their effects on the code cloning, F= 330, p-value= 0.00.

K-Means group and code cloning First, a table was created highlighting the statistics of the considered variables. This information can be observed in Table 8.

K-Means Group	N	Mean	SD	SE	95% Conf. Interval
0	4,989	0.2623	0.2527	0.0036	[0.2553 0.2693]
1	793	0.3271	0.2304	0.0106	[0.3062 0.3480]
2	1,219	0.2986	0.2743	0.0079	[0.2832 0.3141]
3	2,134	0.2540	0.2555	0.0055	[0.2432 0.2648]

Table 8: Relative number of code cloning per assigned K-Means group

The overall average code cloning was 0.2855 (confidence interval 95% with minimum of 0.2720, maximum of 0.2991). This means that on average, a developer clones 29% of its written code. With group averages of 0.2623 95% CI(0.2553, 0.2693) for group 0; 0.3261 95% CI(0.3062, 0.3480) for group 1; 0.2986 95% CI(0.2832, 0.3141) for group 2; and 0.2540 95% CI(0.2432, 0.2648) for group 3.

There is a statistically significant difference between the developer type groups and their effects on the code cloning, F= 73, p-value= 0.00.

5 Discussion

Alternative statistical test An alternative for a statistical test between the continuous and categorical variables would be a one-way ANOVA. To perform a one-way ANOVA, six assumptions have to be validated about the data. The limiting factor why the one-way ANOVA was not performed is because the assumption that the dependent variable should be approximately normally distributed. However, after a quick visual inspection, this assumption could not be validated. Figure 7 highlights this.

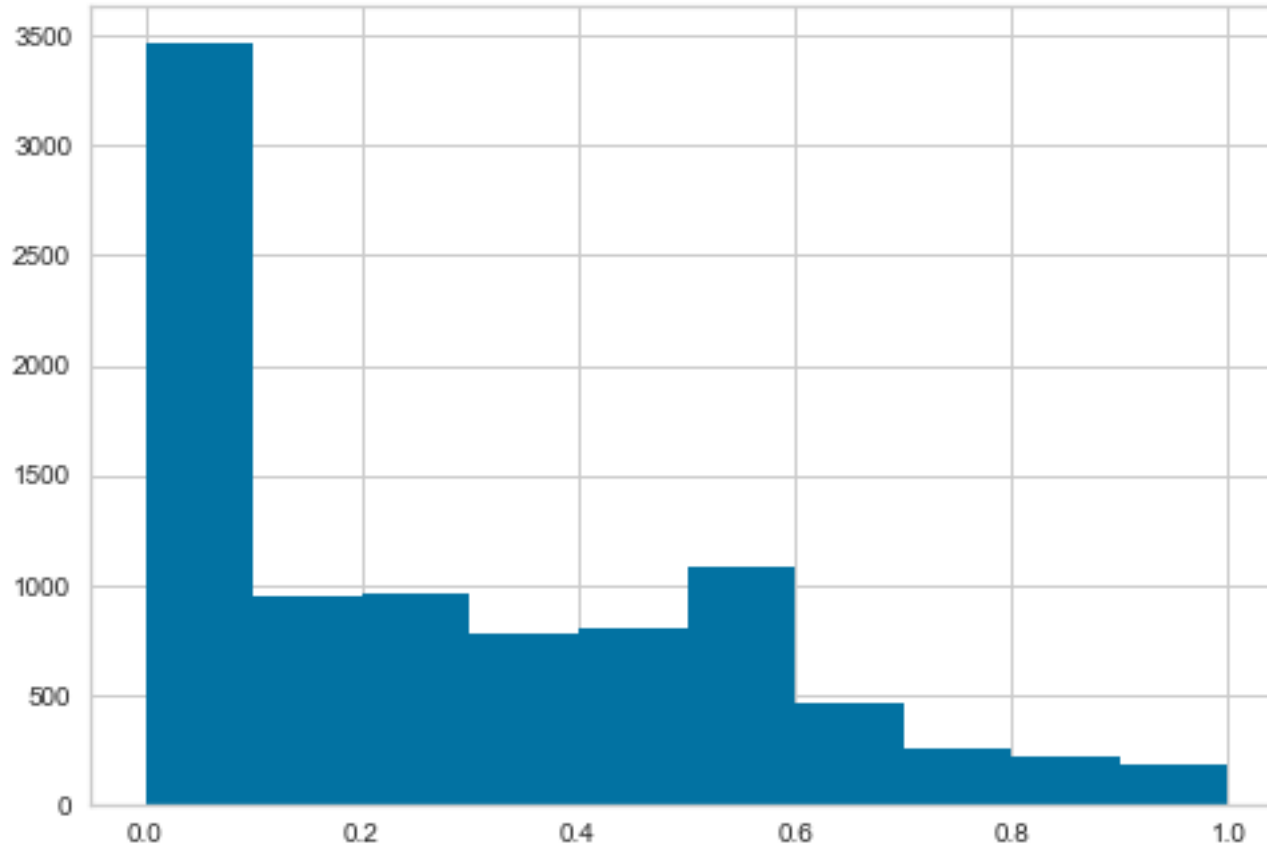

Fig. 7: Bar chart of relative code cloning (methods copied / total methods published) and amount of developers

Limitations For this study, the SearchSECO database was used and the database API had to be used to query the data. This meant only certain requests could be made to the database via TCP. A request always needed some sort of information, like an author or project ID, as opposed to a request where all instances of a certain type are returned based on some condition. Therefore, a list of author IDs was needed with which all methods an author had written could be retrieved. Since this would result in millions of methods retrieved from the database, problems arose with memory. Furthermore, the TCP request did not always return an answer, which meant approximately half of the requested authors were skipped. It is not known whether this problem has to do with the request or the database. This limited the amount of data that could be used.

After creating the initial data set with the methods of many authors, the intention was to request information on every method from the database. This

way it could be established how often a method was cloned and by which authors. But, as the initial data set contained over four million unique methods, similar problems arose in this phase. Again, the TCP request would often return empty answers and the amount of requests and processing needed costed great amount of time and memory. Therefore, it was chosen to only use the initial data set to find duplicates. This gave problems like not knowing for sure which duplicate was written first and not necessarily having all methods of a project in the data set, which is needed when looking at how much a certain developer contributed to a project.

6 Conclusions & future work

This research explored the relation between code cloning and developer types. The main question that guided this research was: *What relationship exists between the number of methods a developer clones and the developer type who introduces these methods in open-source projects?* Further sub questions were *Does the experience of the developer influence the number of code cloned?* and *What types of developers are there within the open-source ecosystem?* The conclusions of this research are as follows: **(1)** there are three different developer types in the open-source software ecosystem, namely Networker, Lone Wolf and One Day Fly, **(2)** Networkers are most prone to code cloning, while One Day Flies are least prone to code cloning, and **(3)** developers with high development experience are most prone to code cloning, while developers with low development experience are least prone to code cloning.

Due to the obvious fact that the SearchSECO database only contains a small fraction of all GitHub projects, it is uncertain whether the results of this research can be generalized. It could be interesting to see whether different projects yield different results; another future research suggestion could be to look into the type of projects, whereas this research did not make any distinction as to what the software projects are used for. Additionally, one could look at different developer types instead of the three presented in this research. Another point of interest is whether the programming language plays a role in code cloning; maybe a developer is more prone to clone code in a particular language than in another language.

References

1. Raihan Al-Ekram, Cory Kapser, Richard Holt, and Michael Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 10–pp. IEEE, 2005.
2. Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How clones are maintained: An empirical study. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 81–90. IEEE, 2007.
3. Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598, 2009.
4. James R Cordy. Comprehending reality-practical barriers to industrial adoption of software maintenance automation. In *11th IEEE International Workshop on Program Comprehension, 2003.*, pages 196–205. IEEE, 2003.
5. Georgina Cosma and Mike Joy. Towards a definition of source-code plagiarism. *IEEE Transactions on Education*, 51(2):195–200, 2008.
6. Daniel M German, Massimiliano Di Penta, Yann-Gael Gueheneuc, and Giuliano Antoniol. Code siblings: Technical and legal implications of copying code between applications. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 81–90. IEEE, 2009.
7. Luqi Guan, John W Castro, Xavier Ferré, and Silvia Teresita Acuña. Copy and paste behavior: A systematic mapping study. In *SEKE*, pages 463–466, 2020.
8. Aram Hovsepyan, Riccardo Scandariato, Wouter Joosen, and James Walden. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th international workshop on Security measurements and metrics*, pages 7–10, 2012.
9. Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *2009 IEEE 31st International Conference on Software Engineering*, pages 485–495. IEEE, 2009.
10. Jaap Kabbedijk and Slinger Jansen. Steering insight: An exploration of the ruby software ecosystem. In *International conference of software business*, pages 44–55. Springer, 2011.
11. Cory J Kapser and Michael W Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.
12. Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614. IEEE, 2017.
13. Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192, 2006.
14. Luana Müller, Milene Selbach Silveira, and Clarisse Sieckenius de Souza. Do i know what my code is" saying"? a study on novice programmers' perceptions of what reused source code may mean. In *Proceedings of the 17th Brazilian Symposium on Human Factors in Computing Systems*, pages 1–10, 2018.
15. Matija Novak. Review of source-code plagiarism detection in academia. In *2016 39th International convention on information and communication technology, electronics and microelectronics (MIPRO)*, pages 796–801. IEEE, 2016.

16. Nam H Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 447–456, 2010.
17. Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. Impact assessment for vulnerabilities in open-source software libraries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 411–420. IEEE, 2015.
18. Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.
19. MA Syakur, BK Khotimah, EMS Rochman, and Budi Dwi Satoto. Integration k-means clustering method and elbow method for identification of the best customer profile cluster. In *IOP conference series: materials science and engineering*, volume 336, page 012017. IOP Publishing, 2018.
20. Synopsys. Open source security and risk analysis report, 2021.

Appendices

A Source code for results

A.1 Applying K-Means algorithm to SECO data

```
from sklearn.cluster import KMeans
from sklearn.cluster import KMeans
from yellowbrick.cluster.elbow import kelbow_visualizer

kelbow_visualizer(KMeans(random_state=3), df_km[['projectContributions', 'methodsWritten']]

kmeanModel = KMeans(n_clusters=5)
kmeanModel.fit(df_km)

df_km['k_means'] = kmeanModel.predict(df_km[['projectContributions', 'methodsWritten']])

df_km['k_means'] = df_km['k_means'].astype("category")
```

A.2 Performing Kruskal-Wallis test on SECO data

$\text{group}_0 = pd.DataFrame.dropna(df_krusk.where(df_krusk['k_means'] == 0))['relativeClones']group_1 = pd.DataFrame.dropna(df_krusk.where(df_krusk['k_means'] == 1))['relativeClones']group_2 = pd.DataFrame.dropna(df_krusk.where(df_krusk['k_means'] == 2))['relativeClones']$

stats.kruskal($\text{group}_0, group_1, group_2$)

$\text{group}_low = pd.DataFrame.dropna(df_krusk.where(df_krusk['developerExperience'] ==' Low'))['relativeClones']group_med = pd.DataFrame.dropna(df_krusk.where(df_krusk['developerExperience'] = Medium'))['relativeClones']group_high = pd.DataFrame.dropna(df_krusk.where(df_krusk['developerExperier High'))['relativeClones']$

stats.kruskal($\text{group}_low, group_med, group_high$)